**Lab 3**

## **Part A: User Environments and Exception Handling**

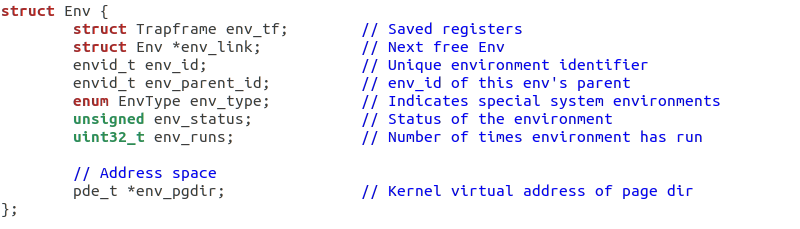
在 kern/env.c 文件中，操作系统一共维护了三个重要的和用户环境相关的全局变量：

struct Env \*envs = NULL; // All environments

struct Env \*curenv = NULL; // The current env

static struct Env \*env\_free\_list; // Free environment list

因此我们看一下Env结构体，这个结构体定义在inc/env.c中，如图1所示

图1

**struct Trapframe env\_tf：**当该用户环境不在运行时保存寄存器的值，当内核从用户态切换到内核态时能够在之后恢复该用户环境；

**struct Env \*env\_link：**指向env\_free\_list中的下一个Env的指针。Env\_free\_list指向列表中的第一个空闲的用户环境；

**envid\_t env\_id：**这个值可以唯一的确定使用这个结构体的用户环境是什么。当这个用户环境终止，内核会把这个结构体分配给另外一个不同的环境，这个新的环境会有不同的env\_id值；

**envid\_t env\_parent\_id：**该用户环境的父用户环境的env\_id；

**enum EnvType env\_type：**用于区别出来某个特定的用户环境。对于大多数环境来说，它的值都是 ENV\_TYPE\_USER；

**unsigned env\_status：**表面该用户环境的状态，有以下几种值

ENV\_FREE：代表这个结构体在空闲状态，应该在链表env\_free\_list中；

ENV\_RUNNABLE: 代表这个结构体对应的用户环境等待在处理机上运行；

　　　 ENV\_RUNNING: 代表这个结构体对应的用户环境正在运行；

ENV\_NOT\_RUNNABLE: 代表这个结构体所代表的是一个活跃的用户环境，但是它不能被调度运行，因为它在等待进程间的通信；

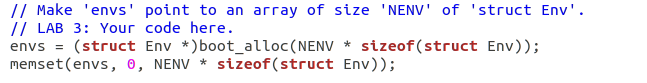
　　　 ENV\_DYING: 代表这个结构体对应的是一个僵尸环境。一个僵尸环境在下一次陷入内核时会被释放回收。

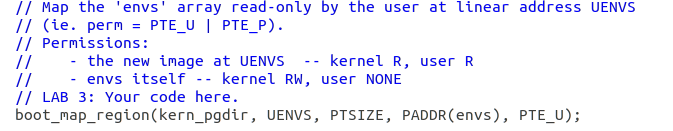
**pde\_t \*env\_pgdir：**存放着这个环境的页目录的虚拟地址

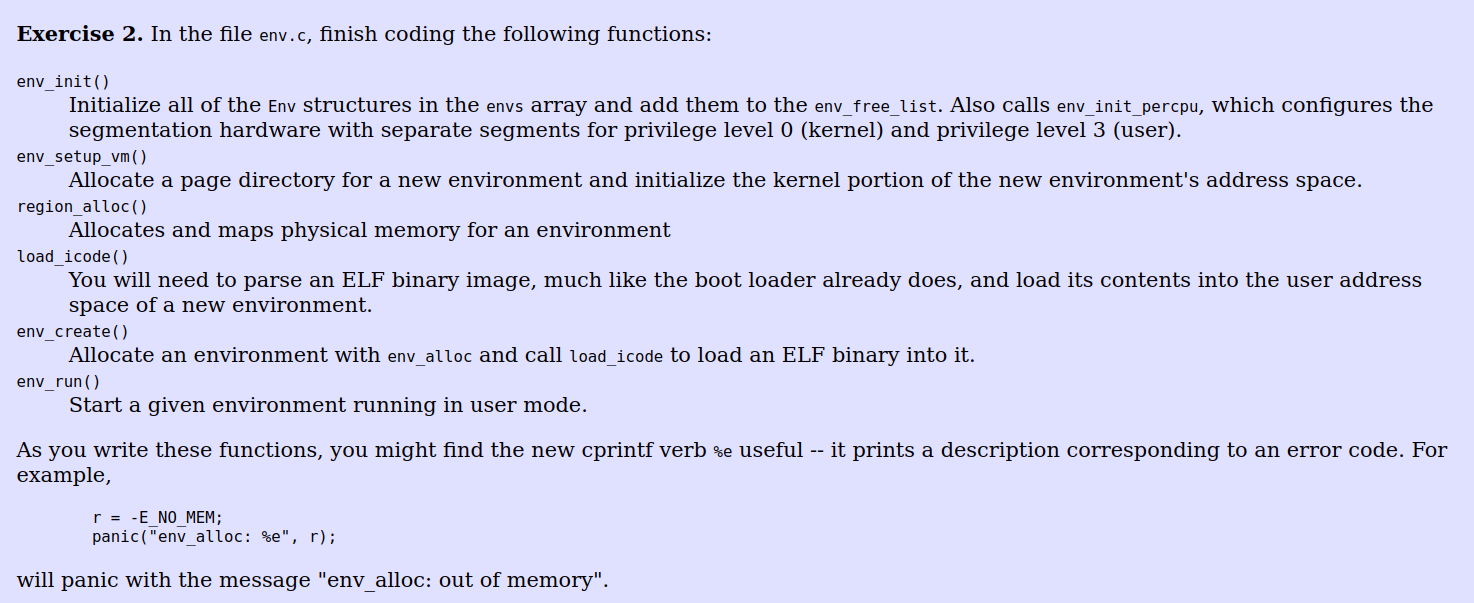
**Exercise 1**. Modify mem\_init() in kern/pmap.c to allocate and map the envs array. This array consists of exactly NENV instances of the Env structure allocated much like how you allocated the pages array. Also like the pages array, the memory backing envs should also be mapped user read-only at UENVS (defined in inc/memlayout.h) so user processes can read from this array.

You should run your code and make sure check\_kern\_pgdir() succeeds.

类似lab2对pages数组的初始化，调用boot\_alloc和memset即可，对于映射，调用boot\_map\_region即可，如图2图3所示

图2

图3

图4

Exercise 2如图4所示，需要实现env.c中的几个函数

**env\_init()**:初始化envs数组中的所有Env结构，并加到env\_free\_list中，然后调用env\_init\_percpu,代码比较容易实现，需要注意的是插入env\_free\_list时的顺序，env\_free\_list需要始终指向下标更小的Env结构，因此循环中下标从NENV-1开始递减，如图5所示

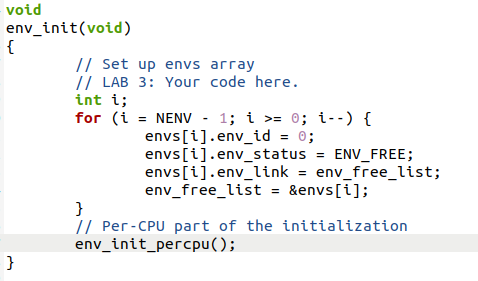


图5

**env\_setup\_vm()**:这个函数主要是用于初始化用户环境的页目录表，但是只初始化跟内核有关的页目录项,不用初始化跟用户环境相关的页目录项。根据提示，所有envs在UTOP之上的虚拟地址都是相同的(除了UVPT)，UTOP之下都为空，并且将kern\_pgdir作为模板，因此代码如图6所示

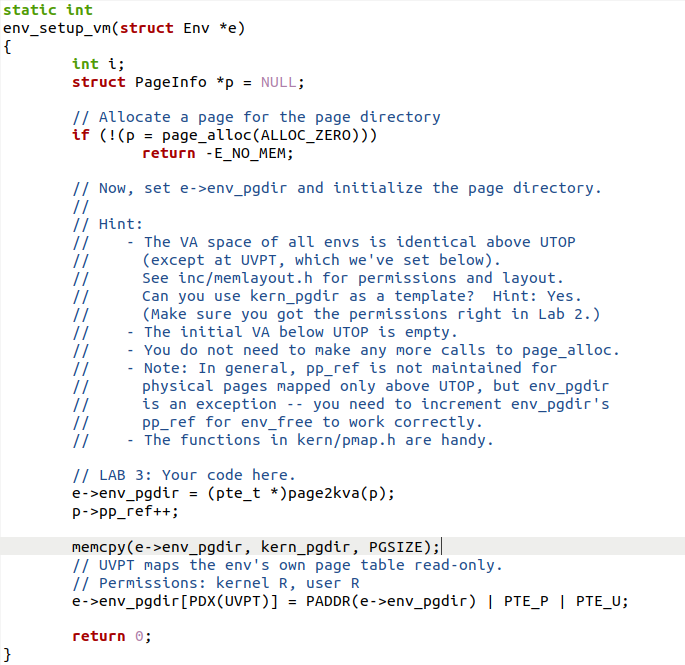


图6

**region\_alloc()**:给用户环境分配len个字节的物理内存，并将其映射到虚拟地址va上，根据提示，地址va向下对齐，va+len向上对齐，权限是内核和用户都可读写，代码如图7所示

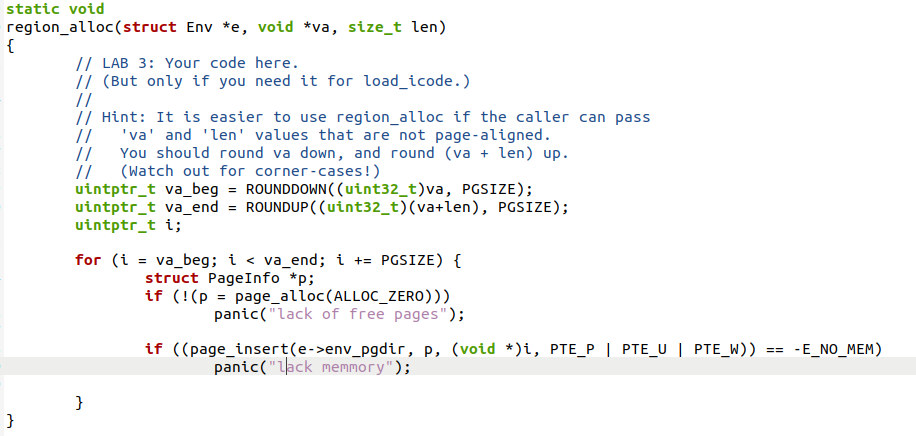


图7

**load\_icode()**:该函数功能是为每一个用户进程设置它的初始代码区，堆栈等等，类似与boot loader的功能，除了boot loader还会从硬盘中读代码。这个函数有一部分是通过网上查询相关答案获得的，主要是因为自己对整个流程不熟悉，对于c语言的语法也不是很熟悉。如图8所示，首先是想不到uint8\_t类型的数据binary能够转化成struct Elf\*类型的数据,其次是要设置程序的执行入口，将header->e\_entry赋给e->env\_tf.tf\_eip，然后是lcr3(PADDR(e->env\_pgdir))这句代码，修改寄存器cr3的值，切换到当前用户环境的页目录下，最后lcr3(PADDR(kern\_pgdir))切换回内核的页目录下

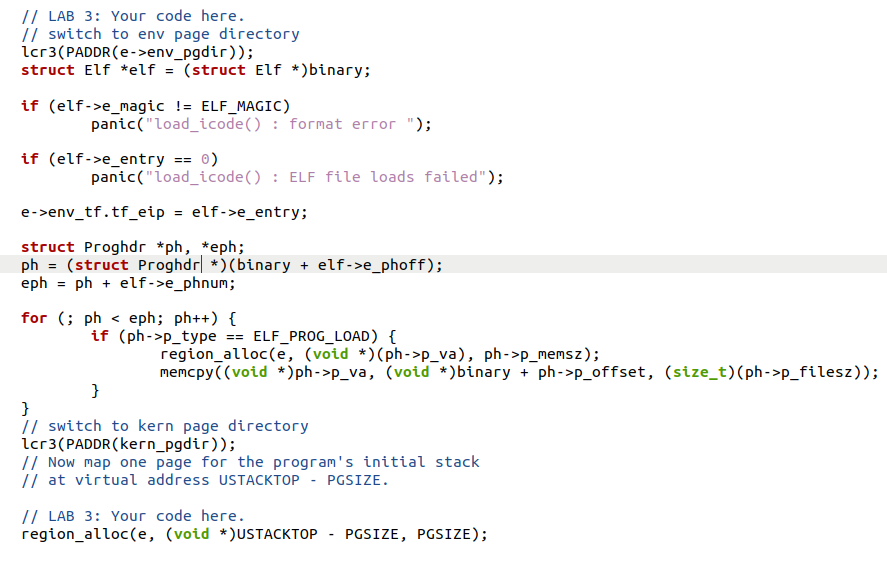


图8

**env\_create():**这个函数实现比较简单，调用env\_alloc函数和load\_icode函数，加载一个ELF文件到用户环境中，如图9所示

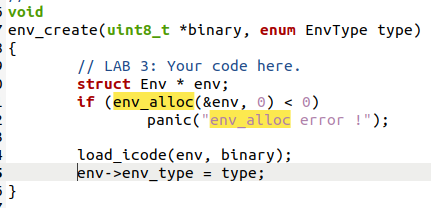


图9

**env\_run()**:这个函数是实现用户环境上下文切换的，根据注释基本上能将代码写出来，如图10所示

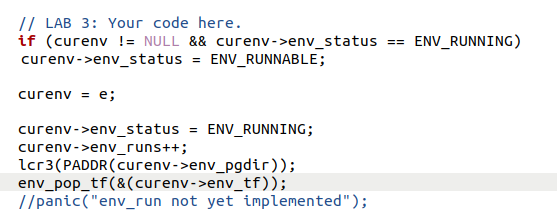


图10

当发生中断或异常时，x86有两种机制来确保使处理器从用户态转移为内核态：

1.**The Interrupt Descriptor Table**

处理器保证中断和异常只能够引起内核进入到一些特定的，被事先定义好的程序入口点，而不是由触发中断的程序来决定中断程序入口点。X86允许多达256个不同的中断和异常，每一个都有唯一的中断向量。一个向量指的就是0到255中的一个数。通过一个表项，处理器可以去加载：

（1）需要加载到EIP寄存器中的值，这个值指向了处理这个中断的中断处理程序的位置。

（2）需要加载到CS寄存器中的值，里面还包含了这个中断处理程序的运行特权级。（在JOS中，所有的异常都是在内核态下处理的，权限级别为0）

2.**The Task State Segment**

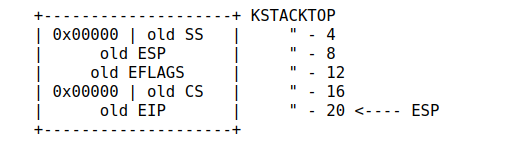
　处理器需要一个地方来存放，当异常或中断发生时处理器的状态，比如EIP和CS寄存器的值。这样的话，中断处理程序一会可以重新返回到原来的程序中。当一个x86处理器要处理一个中断，异常并且使运行特权级从用户态转为内核态时，它也会把它的堆栈切换到内核空间中。一个叫做 “任务状态段（TSS）”的数据结构就用于记录这个堆栈所在的段的段选择子和地址。

### **An Example**

假设处理器在用户环境中执行代码，遇到一个试图除以0的除法指令。

1.处理器切换到由TSS的SS0和ESP0字段定义的堆栈，在JOS中这两个字段分别保存GD\_KD和KSTACKTOP值。

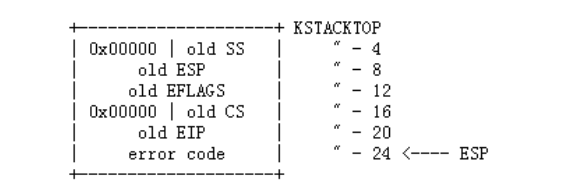
2.处理器将异常参数压入内核栈，从地址KSTACKTOP开始:



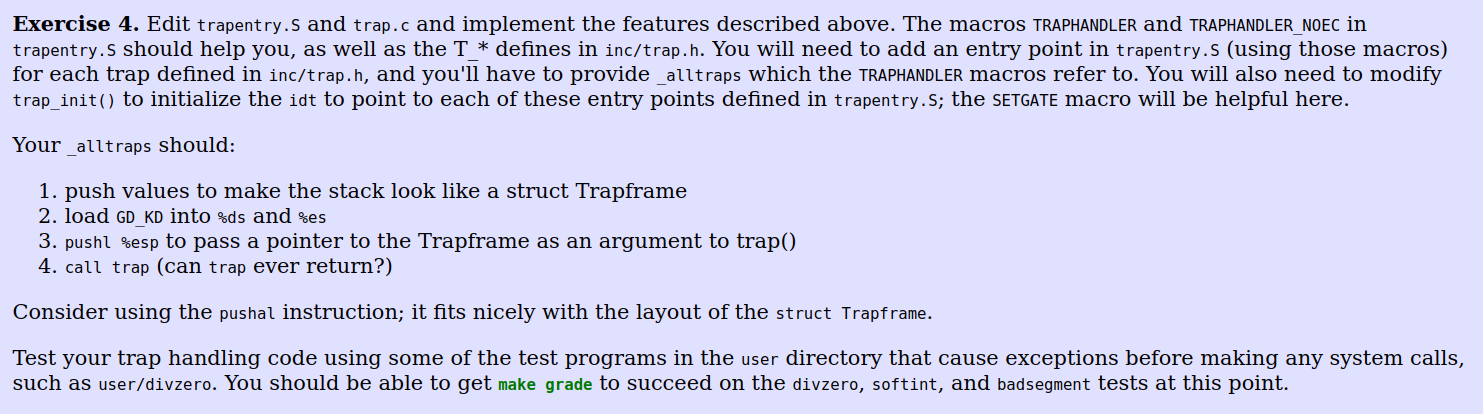
3.因为我们正在处理一个除法错误(在x86上中断向量为0)，处理器读取IDT表中的0号表项，并设置CS:EIP指向该项描述的处理器函数的地址。

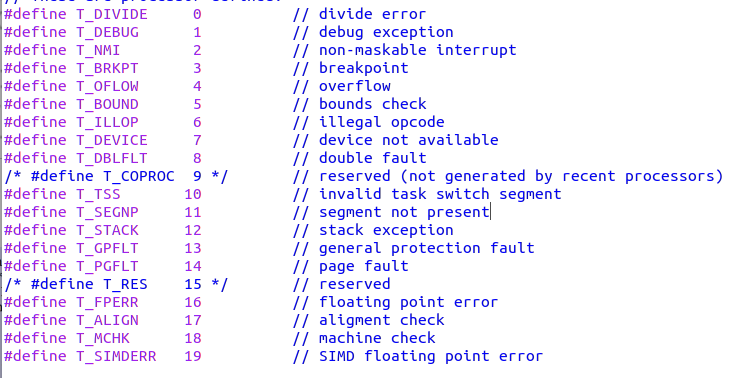
4.处理程序函数获得控制权并处理异常，例如终止用户环境。

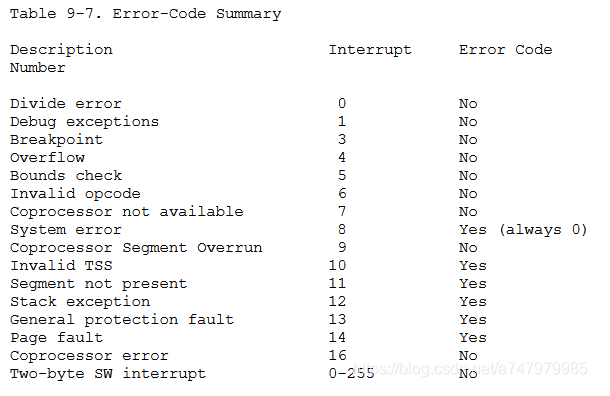
对于某些特定的异常，除了上面图中要保存的五个值之外，还要再压入一个字，叫做错误码。比如页表错，就是其中一个实例。当压入错误码之后，内核堆栈的状态如下：



**Exercise 4**

 首先是实现对trapentry.S的完成，图11是inc/trap.h中定义的中断/异常，trapentry.S中中断需要error code的用宏定义TRAPHANDLER表示，不需要的用TRAPHANDLER\_NOEC，图12是查询资料后找到的相关中断是否需要error code的表

图11

图12

代码如图13所示

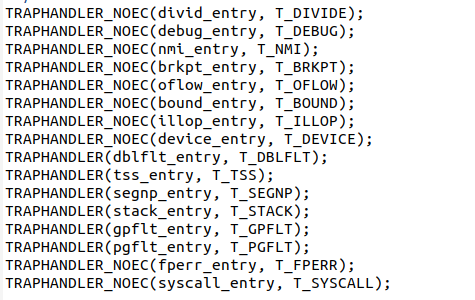
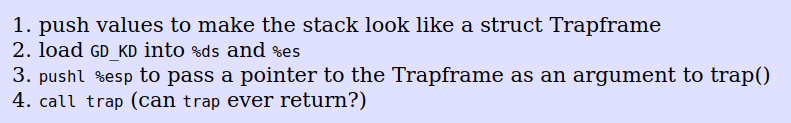


图13

接下来是实现\_alltraps的代码，下面是提示

因此代码如图14所示

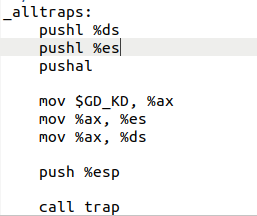


图14

然后是完成kern/trap.c中的trap\_init()函数，初始化idt，使其对应的项指向在trapentry.S中定义的中断，代码如图15所示

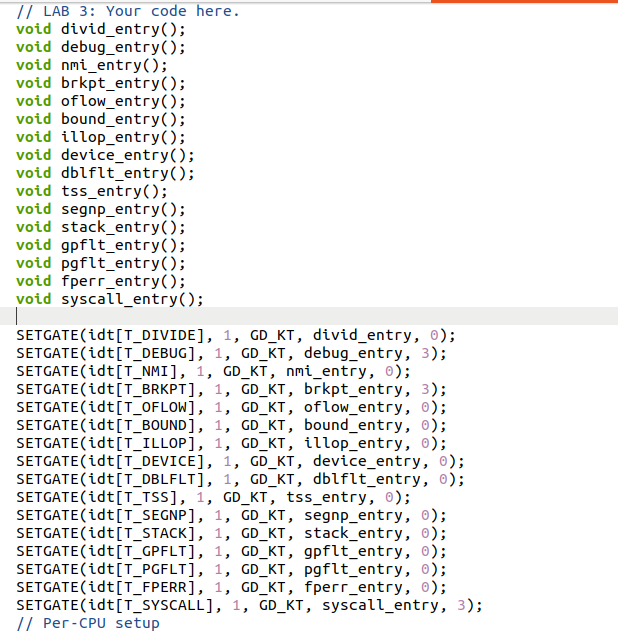


图15

**Questions**

1. What is the purpose of having an individual handler function for each exception/interrupt? (i.e., if all exceptions/interrupts were delivered to the same handler, what feature that exists in the current implementation could not be provided?)

不同的中断或者异常需要不同的中断处理函数，因为不同的异常或者中断可能需要不同的处理方式，比如有些异常是代表指令有错误，则不会返回被中断的命令。而有些中断执行完中断函数还要返回到被中断的程序中继续运行。

2. Did you have to do anything to make the user/softint program behave correctly? The grade script expects it to produce a general protection fault (trap 13), but softint's code says int $14. Why should this produce interrupt vector 13? What happens if the kernel actually allows softint's int $14 instruction to invoke the kernel's page fault handler (which is interrupt vector 14)?

因为当前的系统正在运行在用户态下，而INT指令为系统指令。用户态的程序不能直接调用内核态的程序，会引发一个General Protection falut，即trap 13。如果运行的话可能会导致内核的页表与物理页不匹配

## **Part B: Page Faults, Breakpoints Exceptions, and System Calls**

Exercise 5 和 6是补全trap\_dispatch()函数，对Pages Faults和Breakpoints Exception调用对应的中断处理函数，如图16所示

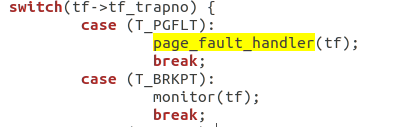


图16

Trapframe结构体中的tf\_trapno表示中断类型，另外，Exercise 5还需要补全page\_fault\_handler函数，如图17所示，寄存器的低两位表示权限级别，如果Trapframe中的cs段寄存器权限级别为0，即为内核态，则说明当前中断（page\_fault）在内核态下发生

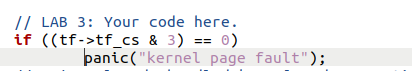


图17

**Questions**

1. The break point test case will either generate a break point exception or a general protection fault depending on how you initialized the break point entry in the IDT (i.e., your call to SETGATE from trap\_init). Why? How do you need to set it up in order to get the breakpoint exception to work as specified above and what incorrect setup would cause it to trigger a general protection fault?

在trap\_init中对breakpoint exception的初始化是**SETGATE(idt[T\_BRKPT], 0, GD\_KT, t\_brkpt, 3);**将最后一个参数dpl设置为3，正常来说会触发breakpoint exception ，如果将其改为0就会触发 general protection fault

1. What do you think is the point of these mechanisms, particularly in light of what the user/softint test program does?

user/softint中的代码是asm volatile("int $14");引起缺页中断，但由于权限不够，会触发 general protection fault

**Exercise 7**

**Add a handler in the kernel for interrupt vector T\_SYSCALL. You will have to edit kern/trapentry.S and kern/trap.c's trap\_init(). You also need to change trap\_dispatch() to handle the system call interrupt by calling syscall() (defined in kern/syscall.c) with the appropriate arguments, and then arranging for the return value to be passed back to the user process in %eax. Finally, you need to implement syscall() in kern/syscall.c. Make sure syscall() returns -E\_INVAL if the system call number is invalid. You should read and understand lib/syscall.c (especially the inline assembly routine) in order to confirm your understanding of the system call interface. Handle all the system calls listed in inc/syscall.h by invoking the corresponding kernel function for each call.**

大致是补全trap\_init()、trap\_dispatch()、syscall()等函数

首先是添加中断向量T\_SYSCALL并注册这个向量，这个类似之前的中断/异常，不再赘述，然后是修改trap\_dispatch()函数使其能够处理system call interrupt，调用 kern/syscall.c中的syscall()函数，并将返回值存储到%eax寄存器中。图18是lib/syscall.c中syscall()函数的代码，其中a、b、c、d表示%eax、%ebx、%ecx、%edx寄存器，D、S表示%edi、%esi寄存器而 kern/syscall.c中的syscall()函数实际上是间接调用了lib/syscall.c的syscall()函数，因此trap\_init()中关于system call interrupt的代码如图19所示

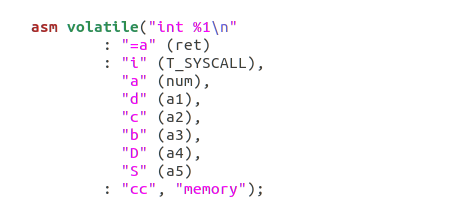


图18

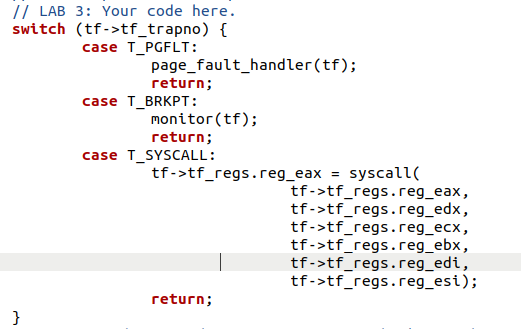


图19

然后是对kern/syscall.c中的syscall()函数的补全，根据参数syscallno的含义，在inc/syscall.h中查找到相关的枚举类型，如图20，因此对syscall()函数的代码如图21所示

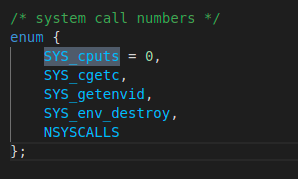


图20

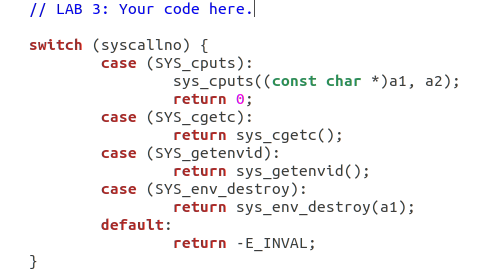


图21

Exercise 8比较简单，完善lib/libmain.c，完成对thisenv的初始化即可，根据提示需要使用sys\_getenvid()，如图22

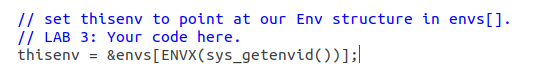
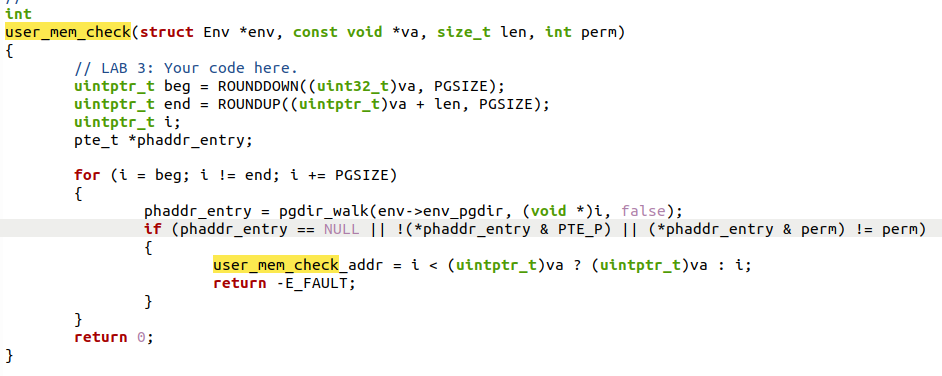
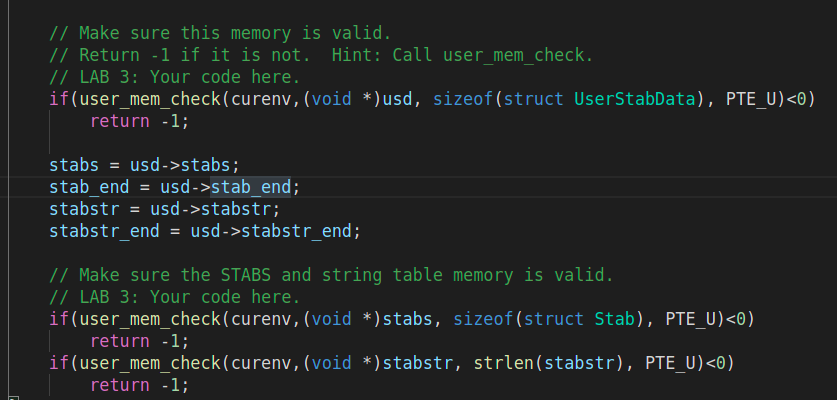


图22

Exercise 9 首先是实现user\_mem\_check函数，这个函数的作用是“Check that an environment is allowed to access the range of memory [va, va+len) with permissions 'perm | PTE\_P'.”，根据代码注释一步一步写出来即可，如图23

图23

然后是在kern/syscall.c中调用user\_mem\_assert函数（这个函数实际上就是user\_mem\_check函数），最后是完善debuginfo\_eip函数，也是调用user\_mem\_check函数，如图24

图24

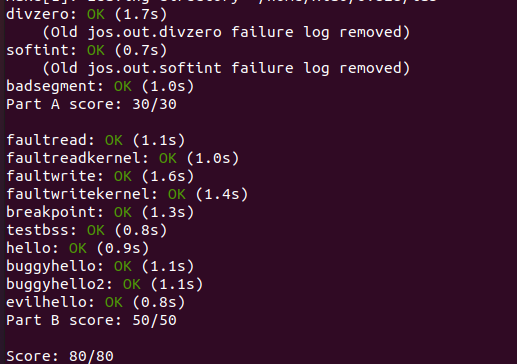


图25

图25是lab3的结果